Application and Evaluation of Built-In-Test (BIT) Techniques in Building Safe Systems

James A. Butler
L-3 Communications

The use of automated functions to monitor and control the activities of potentially hazardous systems is almost unlimited. Ensuring an adequate built-in-test (BIT) (i.e., that a system is fully functional and operational) is one of the most critical aspects of developing safe, automated systems. However, because of its very specialized nature, application, and evaluation, BIT requirements and techniques are often neglected or misunderstood. This article presents some of the goals and uses of BIT, as well as the applications in providing a safe system.

In the early 1960s, the National Aeronautics and Space Administration (NASA) was building the Saturn V spacecraft to go to the moon and back. In order to accomplish this mission, NASA had to develop an automated navigation system. The hardware and software required for this journey was housed in an instrumentation unit (IU). However, no safety-critical software had ever been developed, and the use of computers and sensors to relay safety-critical data had never been used. Digital computers were in their infancy and the hardware was unreliable. In order to compensate, NASA built three identical units, each with its own hardware and sets of input sensors. The outputs from the three units were compared and if one was out of line with the other two, that output was ignored[1].

In today's world, computers are several orders of magnitude more reliable and are much faster. As a result, the use and complexity of tasks assigned to computational systems are also orders of magnitude greater. Today's processors and software programs are used in safety-critical applications, ranging from heart monitors to navigation systems to control of nuclear power plants and weapons. Development of full, independent redundant systems for most of these applications would be cost-prohibitive. As a result, systems and software engineers should consider applying built-in-test (BIT) procedures to the development of all safety-critical functions, determining where failures are likely to occur and what the effect of the failures will be on overall safety, along with providing backup procedures to allow safe task completion.

Although BIT evaluations are common, their use is often limited to a comparatively small number of software and system developers as BIT is not usually taught as a formal curriculum in schools. As a result, BIT may not be considered by systems engineers for new systems, particularly when the purpose of the new system is to demonstrate a new capability or engineering concept. The assumption is that if the design is good and thoroughly tested, all of the hardware/software functions will perform as intended. This article provides the reader with a basic understanding of how BIT techniques are used in embedded systems and how they can be incorporated into new system requirements to reduce the risk of unanticipated system failures. With this understanding, the reader (systems engineers, hardware developers, or software developers) should have at least some capability to evaluate the effectiveness and completeness of BIT requirements and functions employed in safety-critical systems.

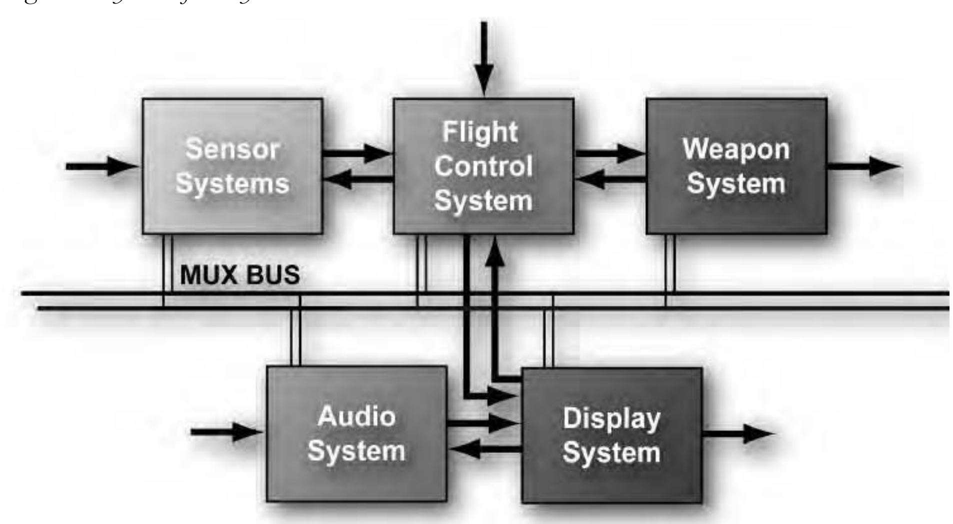
The backup procedures and responses to anticipated errors should also be considered in developing a safe system. The development of backup procedures is unique to each application and is beyond the scope of this article. However, backup sets of inputs provide a partial redundancy at a much lower cost than a full system redundancy. These backup sets of inputs may include additional sensors used to verify the primary sensor inputs. For instance, velocity of an aircraft can be measured by wind velocity or with Global Positioning System (GPS) inputs. The wind velocity is considered more accurate, but the GPS has more checks and may be more reliable; therefore, the GPS inputs may act as the backup for the wind velocity measurements. Alternate procedures, such as activation of a backup system or initiation of manual procedures when the automated system(s) fails, should also be included in the system engineer's requirements.

Typical Embedded System

In order to understand BIT requirements for an embedded system, one has to understand how an embedded system operates. Figure 1 presents a simple system consisting of several subsystems. During normal operation, each subsystem provides and receives data that are critical to performing safety-critical functions. If a sensor system fails to provide the correct data to the flight control system or the data is corrupted during transmission, the safety of the aircraft and passengers could be jeopardized.

Each subsystem may also be comprised of lower-level components, as shown in Figure 2 (see page 16). Each component has the potential of corrupting the subsystem as well as other subsystems in the system. In the example subsystem, Processor 1 reads inputs from var-

Figure 1: *System of Subsystems*



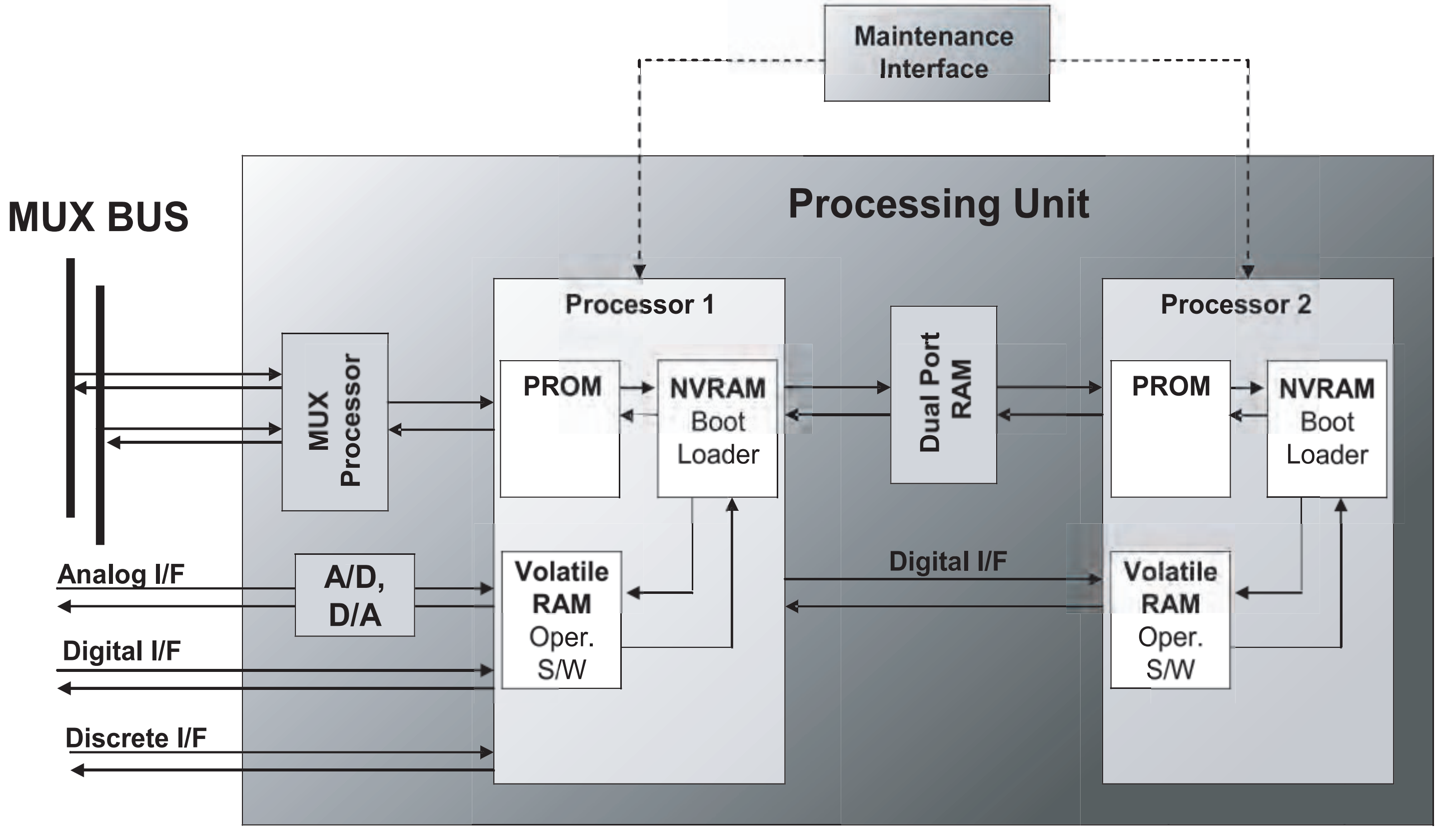


Figure 2: *Typical Embedded System/Subsystem*

ious sensors, user responses, etc; reformats the inputs into engineering units; and provides the data to Processor 2. Processor 2 determines control commands based on the inputs and provides responses to Processor 1. Processor 1 then reformats the outputs and sends the data to each of the respective output units where it may be used on other safety-critical functions.

Interfaces to the system may include operational interfaces such as multiplexer (MUX), digital, discrete, analog, and maintenance interfaces. Internal interfaces include all processor-to-processor and processor-to-peripheral interfaces, including dual-port random-access memory (RAM), which is memory shared by both processors.

The processors typically consist of programmable read-only memory (PROM), non-volatile RAM (NVRAM), and volatile RAM. PROM is used to store data and operational code while the system is not in use. The boot loader is usually installed and runs on NVRAM. During normal startup operations, the boot loader moves the operational code from PROM to volatile RAM and activates it. The volatile RAM is used to house the operational code and provide temporary storage for data during real-time operations.

Software and/or hardware interrupts are usually used to synchronize the activities between processors. In a data-driven system, Processor 1 provides an interrupt to Processor 2 when new data is available. Conversely, when Processor 2 is ready to output its data, it provides an interrupt to Processor 1. In a time-driven system, Processor 1 and/or 2 are provided with time-driven hardware and/or software-driven interrupts.

Operational Functions

In the typical system discussed here, there could be a number of high-level functions including the boot loader, operational code, and interface manipulations:

- **Boot loader.** The boot loader codes typically reside in and operate out of NVRAM. In a multi-processor system, there would probably be a boot loader for each processor. Since the boot loader operates out of NVRAM, the code is usually *burned-in* at the factory. As a result, all reprogramming of the code requires removal of the processor from the board; thus, functional requirements are kept to a minimum. The primary functions of the boot loader include, first, moving the operational code from PROM to RAM and initiating the operational code and, second, providing the capability to read the operational code from the maintenance interface and write it to PROM, thereby providing the capability to update the operational software and providing an access for performing maintenance evaluations on the processor(s).
- **Operational code.** The operational code is responsible for performing the operational functions of the system. After the operational code has been written in RAM, it has full control of the system until the system is powered down or has its authority removed by another system.
- **Interfaces.** The interfaces provide access to data from other systems as well as intra-system data and include MUX, analog to digital (A/D) and digital to analog (D/A), discrete, and dual-port RAM interfaces.

Modes of BIT and Their Functions

Generally, there are four modes of BIT: Startup BIT (SBIT), Continuous BIT (CBIT), Initiated BIT (IBIT), and Maintenance BIT (MBIT).

- **SBIT** is used in the evaluation of key functions and capabilities before the start of the application. SBIT is usually performed or at least initiated by the boot loader and provides a GO/NO-GO response to the system and users. Some of the test functions performed in SBIT include memory, boot load, and interface tests. Startup culminates in initiating the operational code.

- **CBIT** is run out of the operational code and is used to evaluate selected elements and functions during the mission. CBIT is especially applicable to non-expendable systems such as airplanes or weapons, where passenger or bystander safety is involved. The operational code generally performs foreground and background BIT tasks. The foreground tasks include all necessary activities to accomplish the primary operational tasks, including evaluations of inputs to the system. The background tasks include CBIT activities that cannot be performed in the foreground. The following are the activities performed in each:
  - o Foreground tests are the tests needed to assure that the inputs, processor, and software are valid. These tests are preformed in each interrupt cycle. Specific foreground tests include the following:
    - Input tests used in input verification include checksum; parity checks; time tags; sequence numbers; heartbeat checks of digital and discrete inputs and voltage, current, and frequency checks for analog and power inputs.
    - Output tests are used to provide the receiver with the ability to verify the accuracy of the digital message or verify that the analog/discrete outputs were correctly received.
    - Processor evaluations are used to evaluate the health of the processor. Specific tests include interrupt monitoring, evaluation of timing, and memory use.
    - Software evaluations include tests on the software to prevent inaccurate or out-of-bounds data from causing an irreversible error and evaluation of the exception handling procedures provided in some software languages.
  - o Background tests are performed on an *as available* basis. Generally, the background tests require more time to complete than what is available from the system. As a result, background tests are performed after operational code has completed its operational functions. Background tests include the following:
    - Input tests performed in the background including loopback tests of digital, discrete, and analog inputs, as well as non-destructive Stuck-on-1/Stuck-on-0 tests of interface buffers.
    - Memory tests including non-destructive Stuck-on-1/Stuck-on-0 tests of all applicable memory locations.
- IBIT is a detailed BIT used in mission readiness assessments or to assist MBIT in pinpointing sources of errors in the system. Typically, IBIT would be used to pinpoint faults in the system down to the line replaceable unit (LRU) level. IBIT is almost always applicable to non-expendable systems but may also be used in expendable systems such as a *smart weapon*. Since IBIT overrides the operational code functions, it is not used while the system is performing its normal functions.
- MBIT is exhaustive BIT designed to interface with the maintenance port on the unit. A maintenance port is usually provided in the hardware design, specifically to allow for maintenance on the unit. This provides *peek and poke* capabilities into designated memory locations. The more common use is the capability to load new software into the system without having to dismantle the unit or perform any hardware manipulations. MBIT is applicable to all systems as a development platform and a provider of software upgrades.

BITs to Be Performed

Complete evaluation of a system requires several BIT functions. The following is a summary of the BIT functions usually performed in a safety-critical embedded system:

- Download BIT verifies that the software being loaded into PROM from the maintenance port is correct. Download BIT is performed by the boot loader software as part of MBIT.
- Boot Load BIT verifies that the operational software being loaded from PROM to NVRAM is correct. Boot Load BIT is performed in SBIT.
- Memory test verifies that the memory locations (i.e., RAM, scratch pad, working areas) are functioning correctly. Memory tests may be performed in all modes of BIT.
- Interface BIT verifies that the data being passed across the interfaces are correct. Interface tests are used in all modes of BIT but are more critical in CBIT.
- Power BIT verifies that the power inputs are within defined tolerances. As such, Power BIT is used in all modes of BIT.
- Processor BIT verifies the functionality of the processor in real-time. Its primary use is in the CBIT mode.
- Software BIT verifies the operational functionality of the software and is part of CBIT.

BIT Algorithms

There are several ways to perform each of the BITs. The following are some of the more widely used techniques:

- *Checksum* is a sum of all words within a designated memory location and is used to verify inputs to the system or to verify input data matches the data sent or that the data in a memory block has not been corrupted. In performing a checksum, the sum of all words in a message or memory block is written to a designated location. In validating the data block, the data within the block are summed and compared to the original checksum; if the two checksums do not agree, the test fails.
- *Parity* is a single bit sum of all bits within a word or memory location(s) and is used in evaluation of discrete ON/OFF inputs. A typical example is a bank of toggle switches controlling the functions of the software system. An exclusive OR (XOR) is used for the evaluations:

  Parity = Switch_1 XOR Switch_2 XOR . . . XOR Switch_n

  Generally, an odd parity (i.e., XOR an additional 1 to the function) is preferred, as loss of power to the bank of toggle switches is apparent: all zeroes, including the parity, indicate an error such as loss of power to the toggle bank and a 1 indicates that all switches are OFF and valid, and the bank of toggle switches have power.
- *Stuck-on-1/Stuck-on-0* provides the ability to evaluate all bits within a word or memory location to ensure its ability to maintain both 1 and 0. Data patterns that exercise all bits such as 0x55 (01010101) and 0xAA (10101010) are written to the memory location and read back. If the pattern read from the memory location does not match the one written, the test fails. In a non-destructive test, the original value in the memory location is saved and restored once the test is complete.
- *Performance BIT* monitors the processor speed and availability as part of the processor BIT.
- *Time tagging* allows the evaluation of

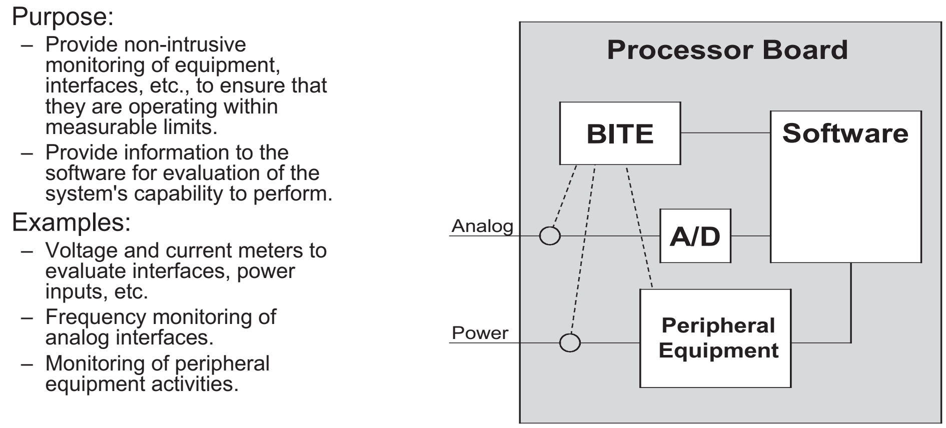


Figure 3: *Examples of Built-In-Test Equipment Functions*

the time that a message was sent or the occurrence of an event with respect to its last occurrence. Time tagging is used most often in validating input messages or as assistance to the receiving processor of another subsystem.

- *Sequence numbering* provides the ability to evaluate input messages with respect to the last message: A 1 is added to the sequence number for each new message by the sender and verified by the receiver.
- *Loop back tests* evaluate interfaces by sending a message to the interfacing processor or analog device which sends the message back. The original message is compared with the message that was looped back.
- *Memory usage* evaluates the available memory during normal use.
- *Interrupt testing* provides evaluation of the interrupt routines and functions.
- *Exception handling* is a pre-programmed reaction to run-time errors. Its applicability in BIT is to assure an adequate reaction to a processor or software error.
- *Stack overflow* is a software test used to evaluate arrayed variables to ensure that they do not overflow the array dimensions and overwrite other data.

BIT Equipment (BITE)

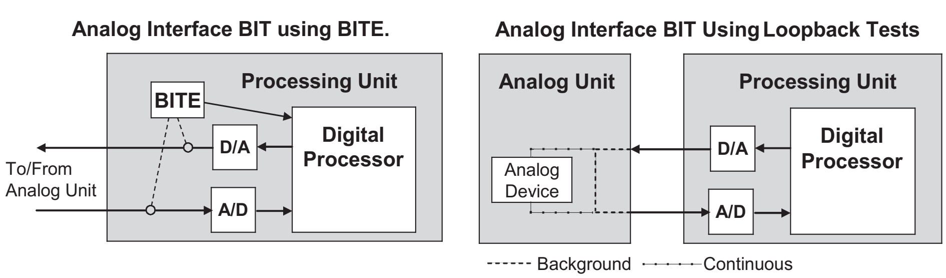
Figure 3 presents an overview of how BITE is used in a processor system. The primary purpose of BITE is to provide a non-intrusive analysis of the systems or interfaces that cannot be directly evaluated using conventional software BIT procedures. The operational code uses the inputs from BITE as a GO/NO-GO indication of the health of the unit or interface being tested.

Typical Test Procedures

Test procedures used in BIT are usually a combination of several BIT techniques or algorithms. The following items present an overview of some of the more common test procedures:

- **Download.** Download of software is typically used to update the operational code stored in PROM. This allows the developer to upgrade the software in the field without having to remove any hardware from the system. The new software is usually read in from the maintenance port and written to PROM. The format for the download code usually includes a header, the text of the code, and a footer. The header includes the number of words in the record, a sequence number for the record, and the function to be performed in the download process. The text contains the operational code and the footer contains the checksum of all words in the record. The procedures used to validate the code include the following:
  - o **Sequence checking.** The *sequence checking* ensures that no record is skipped or read twice.
  - o **Record checksum.** The *record checksum* is used to validate that each record is correctly read and stored in PROM. Each word in the record is written to PROM, read back, and summed with the rest of the words in the record and compared with the checksum in the footer.
  - o **Block checksum.** The *block checksum* is the final download test and is used to make sure that all procedures were done correctly. The final word in a code block is typically a checksum of all words in the block of code. The final validation check is made by re-reading all of the memory in the designated area of PROM and performing a checksum on each word.
- **Boot load.** The procedures for boot load are similar to download; except, time is usually more critical. The data is read from PROM and written to and read back from RAM. A continuous checksum is performed on the data read back from RAM. When all data has been written to RAM, the running checksum is compared with the block checksum in PROM. If the checksums agree, the boot loader initiates and transfers control to the operational code.
- **Memory.** Stuck-on-1/Stuck-on-0 tests are the most widely used methods for evaluating memory. Typically, the boot loader initiates the Stuck-on-1/Stuck-on-0 testing of RAM using a destructive test (i.e., the RAM memory will be cleared before the code is written). In CBIT, non-destructive testing is performed in the background.
- **Interfaces.** Without question, errors introduced in hardware to software and software to hardware interfaces provide the greatest opportunity for introduction of errors into the system. For this reason, the industry has developed some good, and in some cases, sophisticated BIT capabilities. The following presents an overview of the interface BIT capabilities:
  - o **MUX Interfaces.** MUX interfaces, such as the MIL-STD-1553, are purchased as a standard protocol, along with the MUX processor hardware. The 1553 has been around since the early 1970s, primarily because of the BIT used in detecting errors and verifying the data being transferred[2]. Each message received via the 1553 has a header, body, and footer:
    - **Header.** The header provides the sub-address number, number of words in the record,

Figure 4: *Overview of BIT Procedures for Analog Interfaces*



sequence number, and a timestamp.

- **Body.** The body text contains the data being sent to the receiving processor.
- **Footer.** The footer contains a checksum of all data in the record.

The following presents an overview of the tests used by the 1553 to validate the correctness of the data:

- The sequence number is evaluated, ensuring that no messages were lost or that no duplicate was sent.
- The timestamp is compared to the previous timestamp to ensure updates are being received as designed.
- The data is check-summed and compared with the checksum in the footer.
- The MUX BUS provides *heartbeat* messages to indicate it is operational, even when no messages are being sent.
- If any of the previous tests fail, a signal is passed to the operational code so that backup procedures can be activated.

It should be noted that all communication protocols use similar error-checking procedures. Most of the digital audio and visual protocols also contain error correcting routines.

o **Processor-to-processor and dual-port RAM interfaces.** In transferring data from one processor to another, the sending processor writes to a designated memory address used by another processor. Some of the *off-the-shelf* hardware packages such as dual-port RAM have error detecting and error preventing code built into the system. Additional error detecting capabilities are easily included in the BIT process and include sequence numbering, timestamps, heartbeat, and especially checksumming as described in the MUX interfaces.

o **Analog interfaces.** Analog devices are usually very dumb; by their very nature, they do not provide any way of evaluating the input or output signals. Therefore, errors are common from the analog sensor itself or during transmission. Analog interfaces include both inputs and outputs to the system. In order for the digital processor to be able to read analog inputs, an A/D processor converts the analog signal to its digital counterpart. Conversely, the D/A provides the analog device with the electrical signal corresponding to the digital command from the digital processor. Validation of input signals is typically performed by the following:

- **Range checking the signal.** In order for range checking to be effective, the full electrical range of the analog device must not be used for normal inputs. Thus, a *zero* or *full scale* reading is distinguishable as an error and not a normal input.
- **Comparing the signal with redundant analog signals.** The use of redundant signals is a highly reliable method of detecting errors in analog inputs, assuming that all *common cause* errors, such as common grounding, have been eliminated. One way of enhancing the reliability of the redundant signals is to set the signals in opposite directions – sensor A provides inputs from low scale to high and sensor B provides inputs from high to low.
- **Reasonableness checks.** Reasonableness checks evaluate the data to ensure the inputs are reasonable, given the previous data. If the sensor provides inputs that are physically impossible (i.e., the velocity goes from +100 MPH to -100 MPH in a single 0.01 sec. cycle), the sensor data is probably incorrect and should not be used in making critical control commands.

Validation of output signals are usually performed by use of BITE or loopback tests, as presented in Figure 4.

- BITE monitors the analog input and output channels to ensure the correct range of voltage, current, and/or frequency signals are provided.
- BITE can also be used to perform an internal loopback test which can be used to validate the A/D and D/A converters within the processing unit.
- Loopback tests that do not engage the analog device are performed in SBIT or background CBIT.
- Continuous foreground CBIT loopback tests evaluate each signal to the analog device. Since the foreground CBIT evaluates all command signals, it is the preferred method of BIT evaluation for analog outputs.

o **Discrete interfaces.** Discrete interfaces are typically used to activate (turn on) or deactivate (turn off) a particular function or input. Figure 5 provides an overview of testing discrete inputs. Parity evaluations provide evaluation of all inputs at all times and is considered superior when compared to other methods of evaluations, such as loopback tests, which must be performed in the background.

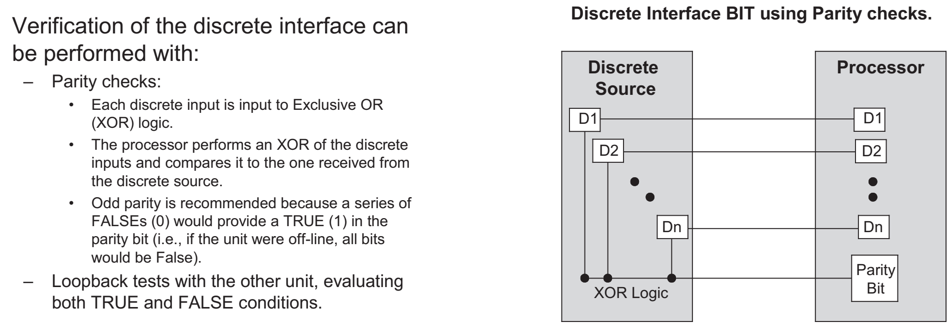


Figure 5: *Evaluation of Discrete Inputs*

- **Processor.** Evaluation of processors is particularly important as new code is added or existing code has been modified. Some of the processor BITs include the following actions:
  - o Evaluate the performance of the system by, first, setting time limits on hardware to software functions and measure the time to perform primary functions; and, second, setting flags to determine if any high-priority functions are not performed in an interrupt cycle.
  - o Measure and compare the memory usage with threshold percentages to ensure adequate capability during operations.
  - o Evaluate timed interrupts by providing hardware interrupts (i.e., watchdog timers) to evaluate software functions and vice versa.
- **Software.** The purpose of software

BITs is not to perform validation of the software but to monitor its functions and inputs to ensure that the software does not crash during critical operations. The procedures available for software evaluations include the following:

- o **Data analysis.** First, perform sanity checks on input data and, second, prevent run-time errors (divide by zero, log of zero or a negative number, etc.) by ensuring that incorrect and out-of-bounds data are not used.
- o **Stack overflow.** Provide software checks to ensure against and report conditions where stacks overflow (especially necessary in C, C++, and other languages).
- o **Exception handling.** Provide exception handling capabilities in the code development so that run-time errors do not cause the system to crash. Ada and other languages provide for exception handling routines.

Confirmation of Emergency Procedures

Providing responses to emergency situations is one of the primary responsibilities of safe software. Ensuring that the indicated response is correct is often dependent on knowing that all inputs, procedures, and commands are correct. To this end, the software system must provide adequate BIT *before* the emergency procedure is activated. The use of foreground rather than background tests reduces the time required to confirm that emergency procedures are required. If errors are detected, the backup or redundant inputs and procedures built into the system provide a safe alternative. If the data was not corrupted, a clear emergency procedure should be activated. If data errors are detected by BIT, the backup or redundant inputs and procedures built into the system provide safe alterations without having to activate emergency procedures.

Summary

The cost of evaluating BIT and implementing recommended improvements is relatively inexpensive when built into the system from the beginning. The ability to perform BIT on most interfaces is determined during the high-level system design; BIT between subsystems requires that both subsystems perform their respective BIT functions. However, correcting interface BIT oversights requires making changes to at least two software components and may also require modification of the hardware design.

There are no hard and fast rules concerning BIT – certainly no one size fits all recommendations. As a result, the application determines the amount and detail of BIT required. The number and type of responses to BIT are dependent on the criticality of the application and the likelihood of the system error. Each response criteria is determined by available software and system backups and the level of operational capability that is desired. As a result, each backup procedure needs to be evaluated for its adequacy in meeting the response criteria.◆

Notes

1. John Duncan presents a great overview of the development of the Instrumentation unit in <www.apollosaturn.com/s5news/p71-7.htm>.
2. Several companies provide very good Web sites to their 1553 products and full standard descriptions are available from the government. The PDF presented, <www.testsystems.com/pdf/overview.pdf>, provides a good overview of the development and use of the 1553.

About the Author

James A. Butler has more than 35 years in software safety analysis and software development, including embedded systems, system engineering, BM/C3, and engineering. He has performed software safety analysis on the Apache helicopter flight management computer. The analyses performed included requirements review, design implementation, and technical adequacy test; analysis of the design using software failure modes, effects, and criticality analysis; and determination of critical software components, inputs, and algorithms. Butler has worked closely with designated engineering representatives in performing software safety evaluations for the Chinook helicopter, digital advanced Flight Control System, and C-130 Intercommunication and Radio System, and he has evaluated applicable safety standards for the Army's Future Combat System.

4121 Hide-A-Way DR
Guntersville, AL 35976
Phone: (256) 505-0808
E-mail: jimjudybutler@bellsouth.net